What is an Array?

- An array is a collection of data items, all having the same data type, and accessed using a common name and an integer index for accessing a particular element of the array.
- Very useful in conjunction with loops, for performing the same actions on all array members without introducing lots of variables.
Array Syntax, by Example

• Declaring an array of 10 ints named “nums”:
  ▪ int nums[10];
• Storing a value in an array element:
  ▪ nums[5] = 42;
• Copying a value from one array element to another:
  ▪ nums[7] = nums[5];

Array Indices in C++ start at 0, and go to one less than the array size:

const int SIZE = 200;
int i, numbers[SIZE];
for( i = 0; i < SIZE; i++ ) {
  numbers[i] = i + 1;
}
// Fills numbers[0]
// to numbers[199]
// with integers 1 to 200
DANGER! DANGER!

“Off-by-One” is a common error

const int SIZE = 200;
int i, numbers[ SIZE ];
for( i = 1; i <= SIZE; i++ ) {
    numbers[ i ] = i;
}
// Leaves numbers[ 0 ] unchanged
// Attempts to write to
// numbers[200], with BAD results

Array Access Mechanism

• The computer starts at the beginning of the array, and then jumps “index” spots away.
• The jump accounts for the size of the data items, i.e. jumping 5 doubles vs. 5 ints or chars
• Providing a bad index could land anywhere:
  – Someone else’s space. (Segmentation Violation.)
  – Your data that you don’t use later.
  – Your data that you do use later.
  – Your code, used later or otherwise.
Implementing a Summation

\[ \text{sumSquares} = \sum_{i=0}^{N} (i + 1) X_i^2 \]

```c
int N = 20;
double sumSquares = 0.0, x[N+1];
// Assume some code fills x
for( i = 0; i <= N; i++ )
    sumSquares += (i+1) * x[i] * x[i];
```

Initializing Arrays

- Uninitialized arrays start with random values.
- Provide initial values in { curly braces }, separated by commas. e.g. = \{ 4, 32, -17 \};
- If an array is fully initialized, the size is optional.
- If an array is sized, but only partially initialized, the remaining elements will be initialized to 0.
- `int nums[3] = \{ 42, 109, -17 \}`, `nums2[] = \{ 1, 4, 9 \};`
- `double errors[100] = \{ 0.0 \}; // All 100 zeros.`
Looping Messages

```c
int i, SIZE = 3, nums[ SIZE ];
string words[ SIZE ] =
{ “first”, “second”, “third” };
for( i = 0; i < 3; i++ ) {
    cout << “Enter “ << words[ i ]
    << “ number: “;
    cin >> nums[ i ];
}
```

Review

Supposing that `nums` has been declared as an array of 10 ints. Which of the following correctly averages the first, fifth, and last elements?

B. `average = (nums[0]+nums[4]+nums[9])/3;`
C. `average = (nums[0]+nums[4]+nums[9])/3.0;`
D. `average = (nums[0]+nums[5]+nums[9])/3.0;`
C++ Allows Declarations After Some Code Has Already Been Executed

• This allows you to find out how big you need an array to be, and then declare it.
• Note, however, that once you declare an array you cannot change the size later.
• In particular, if you use a variable to size an array:
  – The variable must have a positive integer value at the time the array is declared.
  – Changing the value of the variable later does not change the size of the array.
  – Declaring an array inside a loop creates it on the first pass.

Example

```cpp
int i, nWeights;
do {
    cout << “How many weights do you need? “;
    cin >> nWeights;
} while( nWeights <= 0 );
double weights[ nWeights ];
for( i = 0; i < nWeights; i++ ) { // use weights[ i ]
```
Using Multiple ("parallel") Arrays

- Often two or more arrays of the same size are used in conjunction with each other. Example:

```c
double x[nData], sines[nData];
// Assume some code fills x[
for( i = 0; i < nData; i++ ) {
    sines[i] = sin( x[i] );
// Do something with sines
```

Two (or more) Dimensional Arrays

- Arrays of two-dimensions are specified using two sets of brackets: double X[nR][nC];
  - Conventionally the first dimension is rows and the second is columns.
- Higher dimensions simply add more brackets:
  - X[n1][n2][n3][n4];
  - The language does not place a limit on the number of dimensions, but compilers do. (9?)
Initializing Multi-Dimensional Arrays

- Multi-dimensional arrays should be initialized using multiple sets of (nested) braces:
  - `int myData[2][3] = { { 42, 17, -103 }, { 199, -3, 22 } };`

- Spacing and layout shown in this example improves readability for the human – The computer doesn’t care.

- If the inner braces are not included, data fills the first row, then the second, and so on:
  - `int myData[2][3] = { 42, 17, -103, 99, -3, 22 };`

Partially Initializing Multiple-Dimensional Arrays

- Either rows or columns may be partially filled:
  - `int data[5][4] = { { 1, 2, 3 }, { 4, 5 }, { 6, 7, 8, 9 } };`

Uninitialized elements of a partially initialized array are filled with zeros. If inner braces are omitted, the first row is filled, then the second, ..., until all values are used.
Multi-Dimensional Arrays are Commonly Accessed with Nested for Loops

```cpp
for( iR = 0; iR < nRows; iR++ ) {
    for( jC = 0; jC < nCols; jC++ )
        cout << data[ iR ][ jC ] << " ";
    cout << endl;
}
```

- This example uses iR and jC for clarity
- Many programmers use i, j, k, . . . (i.e. $X_{ijk}$)

Review

Which of the following is the correct way to declare an array of ints with 10 rows and 20 columns, initialized to all zeros in C++?

A. `int numbers[ ][ 20 ] = { {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0} };`
B. `int numbers[ 10 ][ 20 ] = { 0 };`
C. `int numbers[ 10 ][ 20 ] = 0;`
D. `int numbers[ 10 ][ 20 ] = 200 * { 0 };`
E. `int numbers[ 9 ][ 19 ] = { 0 };`
Aside: Arrays Versus vector< >s

• Traditional C only supported arrays, of arbitrary dimension. 1-D arrays were often referred to as “vectors”, and multi-D arrays as “matrices”. 2-D arrays are often thought of as “tables”.
• C++ introduced a new data type: vector<type>, (defined in STL, beyond the scope of CS109.)
• The vector< > class template provides many powerful features beyond C-style arrays.

Creating vector< >s

• Syntax: vector<type> variableName( size )
  – Type refers to the type of data stored in the vector.
  – Size may be omitted, creating a zero-length vector.
• Examples:
  – vector<double> inputs( 20 ), outputs(20), temps;
  – vector<int> examScores;
  – vector<string> names;
  – vector<vector<int>> table; // A vector of vectors
Accessing Elements of Vectors

• Note that elements of vectors start at 0 and go to one less than the size, just like C arrays.
• Elements of vectors can be accessed using the C-style [ ]s - inputs[ 5 ] accesses the 6th element.
• Vectors also provide the at( int ) method, which has some benefits:
  – inputs.at( 5 ) = 42; // Equivalent to inputs[ 5 ] = 42;
  – outputs.at( i ) = inputs.at( i ) * scaleFactor;

Valuable vector<> Methods Accessed as variable.method( args )

• size( ) - returns the size of the vector.
• resize( int ) - changes the size of the vector to the size given, either larger or smaller.
• push_back( value ) - Adds “value” to the end of the vector, increasing size by one.
• back( ) - Returns the value at the end of the vector, i.e. the last element.
• pop_back( ) - Removes the last element without returning it, reducing vector size by one.
• See also for more: http://www.cplusplus.com/reference/vector/vector/